Advances in a DSL to Specify Smart Contracts for Application Integration Processes

Eldair F. Dornelles[1], Fernando Parahyba[1], Rafael Z. Frantz[1], Fabricia Roos-Frantz[1], Antonia M. Reina Quintero[2], Carlos Molina-Jiménez[3], José Bocanegra[4], Sandro Sawicki[1]

[1]Unijuí University, Brazil

[2]Department of Language and Systems, University of Seville, Spain

[3]Department of Computer Science and Technology, University of Cambridge, UK

[4]Department of Systems and Computing Engineering, Universidad de los Andes, Colombia

{eldair.dornelles, fernando.parahyba}@sou.unijui.edu.br,

{rzfrantz, frfrantz, sawicki}@unijui.edu.br,

reinaqu@us.es, carlos.molina@cl.cam.ac.uk,

j.bocanegra@uniandes.edu.co

Abstract. *Several languages have been developed for writing smart contracts for specific domains, such as health, finance, and business processes. However, none of them includes the constructors needed for writing smart contracts used in application integration processes which have their own particularities. Such smart contracts are responsible for monitoring the communication between the process and the applications to guarantee that the rights of the parties involved in the integration process are observed. To cover the gap, this paper discusses a domain-specific language for writing smart contracts in the application integration domain. It provides constructors to write smart contracts with a level of abstraction close to the natural language used in conventional contracts.*
Keywords: *Smart Contract, Integration Process, Integration Solution, Domain-specific Language, Enterprise Application Integration.*

1. Introduction

Companies of different segments and sizes rely on applications to support their business activities. Typically, a business environment uses a set of applications that exchange data and share functionalities. We use the term "application" to refer to any software artefact capable of processing data or providing functionality programmatically. To manage the communication between applications in this environment, companies usually use integration processes. An integration process is a software artefact developed using tools provided by the Enterprise Application Integration (EAI) community to orchestrate the integrated applications. Thus, an integration process acts as a middleware that communicates individually with each application, performing specific operations to read, write data, or request or respond to a given function or service.

Usually applications used to support business activities include external applications. Some applications do not belong to the owner of the integration process. Therefore,

it is advisable to monitor the communication between the integration process and the external applications. Monitoring needs to be able to observe a set of constraints. This constrains can be related to service level (e.g., number of requests, hours and days in which access is allowed) or to business level which specify constraints related to the business rules (e.g., required data, type of services or products provided). Current integration platforms do not provide mechanisms for performing this type of monitoring. Therefore, if needed, the monitoring system has to be developed ad hoc as a separate system.

Communication constraints in an integration process can be specified as a smart contract. A smart contract is a self-execution contract that can be written using a general-purpose language (e.g., Java, Javascript, or Python), a Cross-domain language (e.g., Solidity, Michelson, or Plutus), or a domain-specific language (DSL) (e.g., Ergo, DSL4SC or ADICO-Solidity). General-purpose languages do not incorporate natural constructors to write smart contracts; i.e., they only provide generic constructors that make the task unnecessarily cumbersome. Cross-domain languages are languages for writing smart contracts that provide general constructors of contracts. However, they do not have the level of abstraction needed or constructors aimed at specific domains. So, writing smart contracts in these languages requires advanced programming skills. On the other hand, a domain-specific language is a language with a level of abstraction that is adequate for the domain. A DSL incorporates the vocabulary of a specific domain and provides constructors closer to natural language. Thus, writing smart contracts in a DSL is simpler and does not require advanced programming skills.

Recently, [Varela-Vaca and Reina-Quintero 2021] carried out a systematic mapping study of the languages used to write smart contracts. They found several domain-specific languages aimed at distinct domains, such as healthcare, finance, business processes, legal, and others. However, no language was found to support the EAI domain. We have noticed that existing DSLs do not incorporate the essential concepts of EAI, such as application, process, type of operations, and messages. Therefore, in this article, we argue that a DSL aimed at the EAI domain will be welcome by the EAI community. Such a DSL needs to incorporate typical contract constructors explicitly (e.g., contract, clause, type of clause, party, condition) and concepts that are intrinsic to the EAI domain —we highlight them in this paragraph.

In this paper, we present Jabuti, a domain-specific language to write smart contracts in the EAI domain. This DSL has a high-level abstraction, providing constructors close to the natural language used in conventional contracts. Jabuti includes contract concepts (i.e., contract, party, condition, and types of clause) and incorporates the EAI domain's vocabulary (i.e., application, process, type of operations, and messages). These features make Jabuti intuitive. It is easy to learn and produces easy-to-write and read smart contracts. We believe that it will be accessible to non–technical people who will be able to read and possibly write smart contracts. In addition, Jabuti was designed to be platform independent, i.e., a smart contract written using Jabuti can be translated into other languages, such as general-purpose languages: Java, Javascript, or Python; or cross-domain blockchain languages: Solidity, Michelson, or Plutus, to be deployed and activated.

The rest of this paper is organised as follows: Section 2 discusses related work. Section 3 presents the basic concepts of EAI. In Section 4, we present the metamodel of Jabuti. Section 5 presents a case study and a smart contract that has been written in

Jabuti. In addition, we present a snippet of Solidity code that reflects some parts of the code written with Jabuti. Section 6 concludes this paper.

2. Related Work

Since blockchain is receiving substantial mainstream attention from industry and academia, a plethora of smart contract languages have appeared. Thus, in [Varela-Vaca and Reina-Quintero 2021] 101 different smart contract languages were analysed. These languages range from high-level languages, such as Solidity [Ethereum Foundation 2021] to intermediary languages, such as Simplicity [O'Connor 2017], or low-level languages, such as Ethereum Virtual Machine bytecode [Wood 2014]. Furthermore, high-level languages can be general-purpose languages or domain-specific languages. We consider as related work the approaches that propose a high-level domain-specific language for smart contracts.

Existing domain-specific languages can be grouped according to their purpose. The group that is most related to our approach is that of languages designed to specify legal contracts or legal bindings. This group includes CML [Wöhrer and Zdun 2020], Ergo [Accord Project 2019], DSL4SC [Tateishi et al. 2019], Das Contract [Skotnica and Pergl 2019], ADICO-Solidity [Frantz and Nowostawski 2016], SPEC [He et al. 2018], SLCML [Dwivedi et al. 2021] and Symboleo [Sharifi et al. 2020]. In this group, the most related approaches to ours are DSLs that are designed having in mind Event-Condition-Action (ECA) rules, that is, they provide constructors for events, conditions, and actions. CML is the most related to our approach in the sense that is devised as a high-level language that is subsequently translated into Solidity. Ergo is a DSL proposed by the Accord project to specify smart legal contracts. DSL4SC is an intermediate language that is used as a formal model to generate statecharts. This approach uses templates to deal with high-level concepts that are managed in a contract and then transforms these templates into DSL4SC models. Das Contract is a visual DSL that combines concepts of the DEMO methodology [Skotnica et al. 2017] and BPMN [Object Management Group 2014] to model contracts. ADICO-Solidity aims to capture high-level institutional constructs to express rules, regulations, and laws. SPESC is a language whose goal is to collaboratively specify smart contracts in the sense that IT experts, domain practitioners, and business experts collaborate in the specification of the contract. SLCML is an XML-based language for specifying cross-organizational business smart contracts. Finally, Symboleo is a formal specification language for legal contracts whose concepts are based on an ontology that extends the UFO ontology [Griffo et al. 2020]. Although these languages share some concepts with Jabuti such as *Party*, *Obligation*, or *Event*, they are not devised to be applied in an EAI context, and, as a consequence, they can not naturally handle operations that are specific for EAI such as push, pull, and others.

Another group of DSLs includes languages that are devised to specify financial contracts. To this group belong SmartFin [Dean 2019], Egelund-Müller et al. [Egelund-Müller et al. 2017], Findel [Biryukov et al. 2017] and Marlowe [Seijas et al. 2020]. The concepts managed by this group of DSLs are more distant from the concepts managed in the EAI domain. SmaCoNat [Regnath and Steinhorst 2018] is a DSL for expressing smart contracts in natural

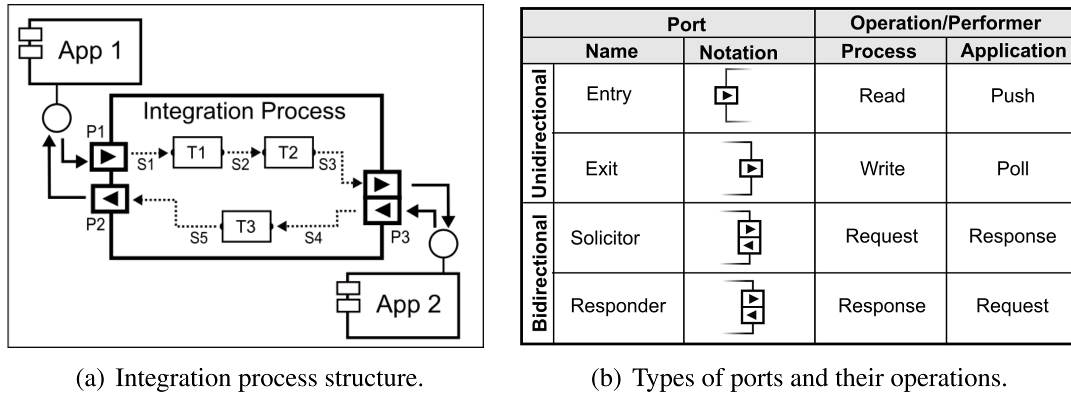
language syntax. Finally, there are approaches to specifying smart contracts in other domains, such as supply chain [Bistarelli et al. 2021]. None of the previous DSLs can be used directly in the EAI domain.

3. Background

In this section, we introduce some basic concepts of the enterprise application integration and smart contracts, which are required to understand our proposal.

3.1. Enterprise Application Integration

Enterprise application integration is a field of software engineering concerned with developing methodologies, techniques, and tools for designing integration processes. An integration process is a software artefact that acts as a middleware orchestrating the communication between integrated applications. Integration platforms are used to design integration processes. An integration platform is a tool that provides the needed resources to design, implement, and execute integration processes. Figure 1(a) shows an integration process that is composed of two applications (App 1, App 2), a set of tasks (T1, T2, T3), slots (S1, S2, S3, S4, S5) and communication ports (P1, P2, P3). Tasks perform atomic actions on messages, such as split, filter, translate, and merge. Through these actions, tasks transform the message data, get the relevant content, and translate them into the format of the target application. Slots are buffers that connect a task to another task and to ports and help to desynchronize tasks. Ports are mechanisms that implement low-level protocols that enable the integration process to interact with their integrated applications. Ports can be unidirectional or bidirectional, as shown in Figure 1(b).



	Port		Operation/Performer	
	Name	Notation	Process	Application
Unidirectional	Entry		Read	Push
	Exit		Write	Poll
Bidirectional	Solicitor		Request	Response
	Responder		Response	Request

(a) Integration process structure.

(b) Types of ports and their operations.

Figure 1. Integration Process: Basic concepts.

There are two types of unidirectional ports, namely entry and exit ports. An entry port is used to input data into the integration process. Ports support read and push operations. Read is performed by the integration process to read data from its applications. Push is performed by an application to input data into its integration process. On the other hand, an exit port is used by an integration process to input data into an application. Exit ports, support write, and poll operations. Write is performed by the integration process to input data into an application. Poll is performed by an application to consume data that have been processed by the integration process. There are two types of bidirectional ports: solicitor and responder ports. A solicitor port is used by the integration process

to perform a request against an application. Solicitor ports are used by the integration process to perform a request operation to request data or services against the application. The application performs a response operation to provide the data requested by the integration process. On the other hand, a responder port is used by the application to perform requests to the integration processes. The application uses a responder port to perform a request operation to request data from the integration process; next, the latter performs a response operation to provide the requested data.

3.2. Smart Contracts

Smart contracts are self-executing digital contracts that aim to ensure that agreements entered between two or more parties are fulfilled [Szabo 1997]. That is, smart contracts are computer programs that define the rules and consequences in the same way as a traditional contract, defining the obligations, rights, prohibitions, and the possible penalties to the involved parties. In addition, smart contracts make it possible to automate the execution of contracts and allow the parties to observe all execution, i.e., which clauses were executed and which of them it lake to execute yet.

A smart contract is made up of a set of attributes, which represent the general properties of the contract, such as the duration of the contract, the parties involved, the asset being traded, and a set of clauses. There are many ways to model smart contracts: financial, technical, and business, among others. One of the most used is the one with a focus on business, as the words attributed and used are closer to the knowledge of business people and therefore more friendly to use. The words attributed to this model are presented below:

- **Contract**: Represents an agreement between two or more parties. It is composed of a set of clauses which arbitrate whether or not a specific action can be performed.
- **Party**: A party represents a natural or legal person involved in the agreement signed in the contract.
- **Action**: An action represents an operation that can be performed by one of the parties. The execution of one action results in one event, which defines the next step in executing the contract.
- **Condition**: A condition is part of a clause; it defines a rule which must be met so that a specific action can or cannot be performed.
- **Clause**: A clause establishes a relationship between a part, a condition, an action, and possible penalties. Clauses can express a right, an obligation or a prohibition.
  - **Right**: A right clause sets out some conditions under which one of the parties can perform a certain action.
  - **Obligation**: An obligation clause determines the conditions under which one of the parties is obligated to perform a certain action.
  - **Prohibition**: A prohibition clause defines the conditions under which one of the parties cannot perform a certain action.
- **Event**: An event is a result generated by the execution of an action. That is, whenever an action is executed, an event is generated. This event can trigger another clause, a penalty or terminate the contract.

In the enterprise application integration domain, smart contracts can be used to define the clauses that express the conditions under which the operations described in the subsection 3.1 may or may not be performed. In our approach, we associate a smart contract with each port. The contract defines the clauses that govern the communication between the integration process and a given application.

4. The DSL Metamodel

In this Section, we present the metamodel of Jabuti and describe its elements. Figure 2 shows the metaclasses and the relationships between them in the metamodel, as well as the properties and operations on each metaclass. Jabuti provides constructors that allows us to model contract concepts, such as `Contract`, `Party`, and `Clause`, as well as concepts that are specific to the application integration domain, i.e. the parties involved in the contract (`Process` and `Application`) and the types of `Operations` that are carried out by ports (`PUSH`, `POLL`, `WRITE`, `READ`, `REQUEST`, `RESPONSE`).

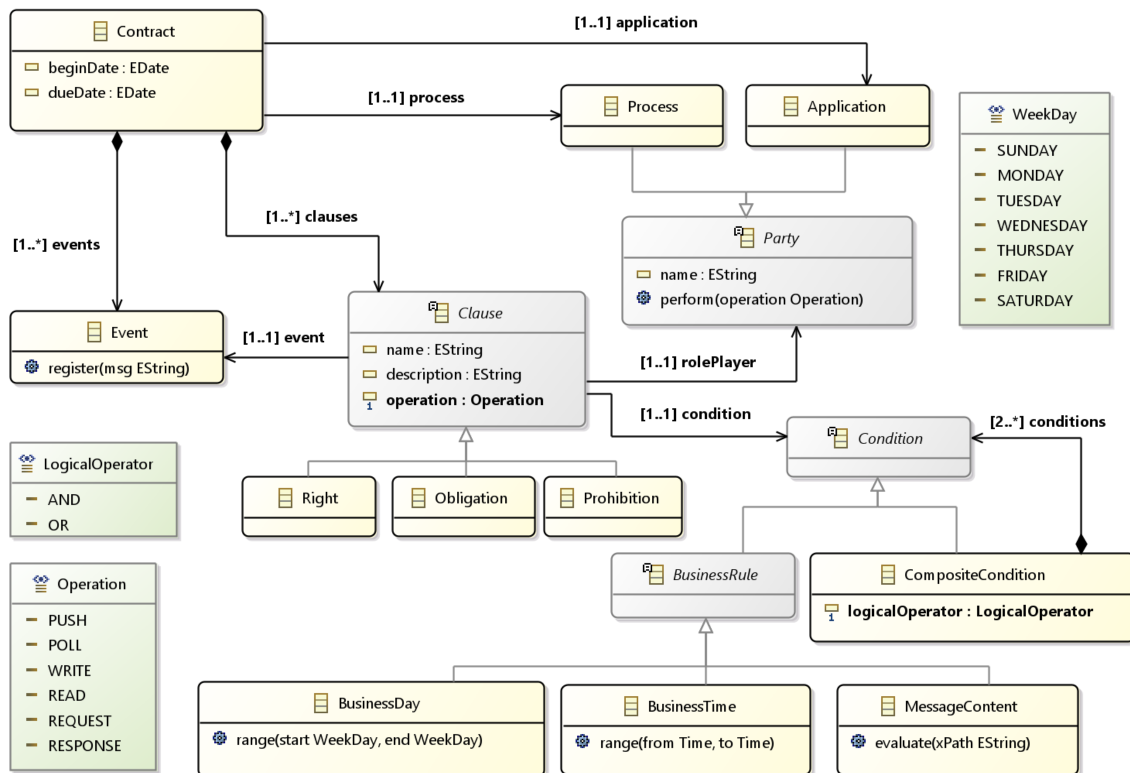


Figure 2. Jabuti metamodel.

The **Contract** metaclass defines the scope of the contract. A contract involves two parties, a `Process` and an `Application`, one or more `Clauses`, and one or more `Events` generated by the contract execution. In addition, the contract contains a start and expiration dates (`beginDate` and `dueDate`).

The **Party** metaclass is used to represent the parties involved in the contract. A party can be a `Process` that represents the integration process or an `Application` that represents the integrated application. The `perform` method defined in this metaclass

represents the action performed by the parties during the execution of the integration process. The parties are associated with the clauses. Each clause declares an active party. This active party is defined by the property `rolePlayer` and indicates which party will trigger the clause.

The **Clause** metaclass is used to specify the condition under which the rolePlayer will be able or not to perform the operation associated with the clause. The operation property can assume the values: push, poll, write, read, request, or response, and represents the operations executed at the port during the execution of the integration process. Clauses can be of the following three types: `Right`, `Obligation` or `Prohibition`. A `Right` clause defines the right of the `rolePlayer` to perform an operation. An `Obligation` clause specifies that the `rolePlayer` must perform the operation specified in the clause. The `Prohibition` clause defines the condition under which the rolePlayer is not allowed to perform the operation specified in the clause.

The **Condition** metaclass is used to define the service-level and business-level rules. The service-level rules are modelled by the metaclasses `BusinessDay` and `BusinessTime`, which have the `range` method that is used to specify the period in which a given operation can be performed. The business-level rules are modelled by the `MessageContent` metaclass, which specifies the conditions related to the business domain. The `MessageContent` metaclass has an evaluation method, which receives an XPath expression (String), which is used to query the content of the received message (XML format) and check whether the business rule is satisfied or not.

The **Event** metaclass is used to model messages with details on the execution of a clause. When a clause calls an event, the event collects some data of this clause, such as name, clause type, rolePlayer, description, execution time, and execution status, to generate the message. In addition, the message generated by the event may contain custom text that the contract designer should provide. Finally, the **Event** metaclass logs the message in the repository using the `register` method.

5. Example of Application

Our study case is based on a real problem (simplified in this paper) involved in the docking process of ships in the port of Seville, Spain. Before docking at the port, the maritime agent (the ship) must request permission to dock from the port authority. This permission is requested by sending an e-mail to the port authority. This e-mail must specify the intended arrival and departure dates and time, the name of the ship, the tonnage of the ship, the Maritime Mobile Service Identity (MMSI) [1], the International Maritime Organization (IMO) [2], and the required port services. In this study case, we consider that the ships can not dock at the port on Sundays and from 10 p.m. to 8 a.m. on the other weekdays. Upon receiving the docking request, the port authority registers it and checks if a dock could be assigned to the ship during the requested dates. A confirmation or denying response is sent by e-mail back by the port authority to the maritime agent. If the e-mail sent is a confirmation, then it should include a proposal with the total cost. This proposal will be valid for 60 minutes. If there is no available berth for those dates, the response includes available dates. If the maritime agent accepts the port authority

[1] MMSI is an unique identifier of a ship station for telecommunication purposes
[2] The IMO number is a unique identifier for ships

proposal and budget, she sends within 60 minutes another e-mail requesting the booking, which includes the identification number of the received proposal. Once the reservation request is received, the port authority registers the reservation and sends a confirmation message by e-mail to the maritime agent.

5.1. The Integration Process

We have used Guaraná DSL [Frantz et al. 2011, Frantz et al. 2016, Frantz et al. 2020] to design a solution to support and automate the docking process of ships introduced in our study case. Figure 3 shows the integration process that we have used, which orchestrates the e-mail server of the port authority and its berthing application.

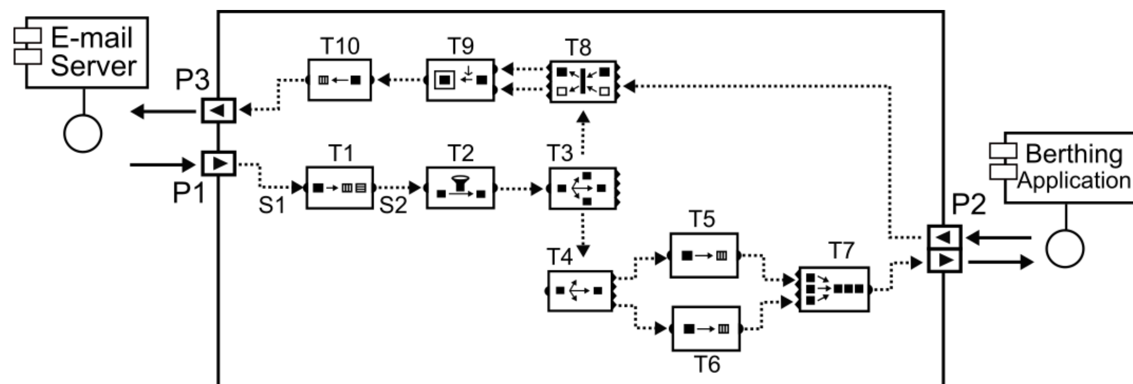


Figure 3. The berthing integration process.

Communication between the e-mail server and the integration process is carried out through the entry port P1 and exit port P3, which are responsible for reading and writing e-mails from and to the server, respectively. For every read operation performed by port P1 in an interval of 15 minutes, a new message is introduced in slot S1. Because messages at this point may contain data from more than one e-mail, the splitter task (T1) splits a message into two or more messages and writes them to slot S2, so that each message carries only the data of a single e-mail. The e-mails could be about several subjects. Due to this, the filter task (T2) filters these messages and keeps in the workflow only the messages related to the berthing requests or booking confirmations. The messages that remain in the workflow are replicated in the replicator task (T3) and forwarded to the dispatcher task (T4) and the correlator task (T8). Task T4 checks the content of the messages and identifies the subject. Messages related to the berthing requests are dispatched to the translator task (T5). The messages related to booking confirmation are dispatched to the translator task (T6). Task T5 creates a berth permission request message based on the received e-mail content from task T4. On the other hand, task T6 uses the content of the e-mail to build a request message to confirm the berth booking. The request messages created by the tasks T5 and T6 are forwarded to the merger task (T7), which gathers these messages and routes them to port P2.

Each message that arrives at port P2, produces a request to the Berthing Application. The application responds to the port either confirming or denying and providing a budget for docking the ship or confirming the booking to the maritime agent, according to the message request type received. Task T8 correlates this response with the original

message containing the data of the e-mail received from the maritime agent. Task T9 enriches the original message with the response, and task T10 transforms the content of this message into text in natural language to be wrapped in an e-mail and sent using port P3 as a formal response to the maritime agent.

5.2. Clauses

Taking into account the description of the study case, we have defined five clauses to monitor the communication between the integration process and the Berthing Application, which occurs through the port P2. In this paper, we present only the smart contract associated with the port P2.

Clause 1: The integration process has the `right` to perform a berthing request operation against the Berthing Application. This request is performed under the following condition: the request must include the arrival and departure dates and times, the name of the ship, the tonnage of the ship, the MMSI, the IMO, and a list of the required services.

Clause 2: The Berthing Application has the `obligation` to perform a response operation against the integration process with a berthing proposal. This response is performed under the following conditions: the Berthing Application must respond within 60 minutes, and must include the berthing dates and budget.

Clause 3: If the maritime agent accepts the proposed berthing conditions, the integration process has the `right` to perform a request operation to book the berthing dates. This request is performed under the following conditions: the request must respond to the Berthing Application within 60 minutes, and must include the identification number of the proposal received.

Clause 4: The Berthing Application has the `obligation` to perform a response operation to the integration process with the booking confirmation. This response is performed under the following conditions: the Berthing Application must respond within 60 minutes, and must include the identification number of the berthing proposal.

5.3. Contract in the DSL

In this section, we introduce a smart contract developed using Jabuti to monitor the interaction between solicitor port P2 of the integration process designed in our study case and the integrated Berthing Application (cf. Figure 3).

The queries to dock the ship received by the integration process at port P1 give place to the execution of request operations at port P2 so that the Berthing Application has to analyse the request and respond to the port with a message denying or containing a proposal with the date, the time, and the cost to dock the ship at the port. Responses are delivered by e-mail to the maritime agent via port P3. Listing 1 presents the smart contract that we have implemented in our editor. This contract is composed of a set of attributes, the implementation of the four clauses introduced in our study case, and an event to register eventual failures in the execution of any clause.

The attributes (Atr) on lines 2 to 5 declare the basic and general information of the contract, such as the start and end dates and time of the contract, as well as the parties involved in this contract. The first clause, (C_1), declares a `right` of the integration

```
Contract BerthingAndServicesRequest{
    beginDate = "2022-01-28 12:00:00"
    dueDate =    "2023-01-27 11:59:59"
    application = "Berthing Application"
    process =     "Integration Process"

    Right requestBerthing {
        description = "Required data to be able to request berthing."
        rolePlayer = process
        rolePlayer.perform(Operation.REQUEST)
        condition {
            MessageContent.evaluate('//arriveData != ""') AND
            MessageContent.evaluate('//departureData != ""') AND
            MessageContent.evaluate('//MMSI != ""') AND
            MessageContent.evaluate('//IMO != ""')AND
            MessageContent.evaluate('//portServices!=""')
        }
        onBreach(registerFailing, "Error sending berthing request.")
    }

    Obligation respondToBerthingRequest{
        description = "Timeout for responding and required data."
        rolePlayer = application
        rolePlayer.perform(Operation.RESPONSE)
        condition {
            Timeout(3600) AND
            MessageContent.evaluate('//arriveData!=""') AND
            MessageContent.evaluate('//departureData!=""') AND
            MessageContent.evaluate('//portServices!=""') AND
            MessageContent.evaluate('//budget!=""') AND
            MessageContent.evaluate('//proposalID!=""')
        }
        onBreach(registerFailing, "Error responding to berthing request.")
    }

    Right requestTheBookingOfBerthingDates{
        description = "Timeout to respond to the port proposal."
        rolePlayer = process
        rolePlayer.perform(Operation.REQUEST)
        condition {
            Timeout(3600)AND
            MessageContent.evaluate('//proposalID!=""')
        }
        onBreach(registerFailing, "Error requesting the booking dates.")
    }

    Obligation respondConfirmingTheBerthingBooking{
        description = "Timeout to confirm the booking has been made."
        rolePlayer = application
        rolePlayer.perform(Operation.RESPONSE)
        condition {
            Timeout(3600) AND
            MessageContent.evaluate('//proposalID!=""')
        }
        onBreach(registerFailing, "Error confirming booking has been made.")
    }

    Event registerFailing(msg : String) {
        register(msg)
    }
}
```

Listing 1. Berthing smart contract in Jabuti.

process to execute a request against the integrated application when the conditions (R) of this clause are met. If this clause is breached, then the method `onBreach`, in line 18, is executed, and a message detailing the breach is forwarded to event Ev in line 58.

Clause C_2 declares the `obligation` of the application to perform a response

operation by sending a berthing proposal to the integration process, and the condition that must be met to perform this operation. Clause C_3 declares the `right` of the integration process to perform a request operation to book the berthing dates when the conditions of this clause are met. Clause C_4 declares an `obligation` of the application to execute a response operation to the integration process to confirm the booking of the berthing dates. Event Ev at line 58 invokes the `register(msg)` method, which is responsible for registering the message log regarding the breach of the clause.

5.4. Contract in Solidity

Smart contracts gained more prominence with the emergence of blockchain technology which provides several services including indelible data, reliability, transparency and consensus without the involvement of central authorities. Currently, there are several blockchain platforms and smart-contract languages, Solidity is one of the best known. It is a cross-domain language that allows the specification of smart-contracts that can be run on Ethereum, one of the leading blockchain platforms. Solidity is supported by a development environment that help programmers to write, test, and deploy smart contracts. It has been used in a wide variety of domains. In this section, we map the Jabuti constructors into their equivalents in Solidity (Table 1). `Application` and `Process` constructors are mapped into addresses. `Clause` is mapped into Function. `BusinessRule` is mapped into require. `Operation` and `WeekDay` are mapped into enum, and `Event` is mapped into event.

Table 1. Mapping of Constructors from Jabuti into Solidity.

Jabuti	Solidity
Application, Process	address
Clause	function
Operation, WeekDay	enum
BusinessRule	require
Event	event

The equivalent Solidity code of part of the berthing smart contract written using Jabuti, (see Listing 1), was hand-coded and is shown in a snippet in Listing 2. The essential attributes of our contract are declared using the Solidity constructors in the snippet from line 8 to line 11 (Atr). Function C_1 in this snippet corresponds to clause C_1 in Listing 1. The conditions of clause C_1 in the contract are represented by the requirements (R) from line 27 to 31 in the snippet.

The algorithms shown in Listing 1 and 2 demonstrate the high level of abstraction provided by Jabuti in comparison to Solidity. Jabuti provides constructors that are frequently used in contract of the EAI domain, such as the `Application` or `Process` and `Clause`. These constructors are close to natural language as opposed to their corresponding constructors (`address` and `function`) in Solidity. Furthermore, the executable smart contract code in Solidity (or any other cross-domain language) can be automatically generated from Jabuti using model-driven engineering text-to-text transformations. This turns smart contracts written in Jabuti into blockchain platform independent code. Automatic transformation minimises coding errors, such as integer overflow and

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.7;
pragma experimental ABIEncoderV2;

contract BerthingAndServicesRequest{
    uint beginDate;
    uint dueDate;
    address process;
    address application;

    RequestForm[] requestList;
    ResponseForm[] proposalList;
    uint[] approvedProposalId;
    uint requestCounter;
    uint proposalCounter;

    ...

    constructor(address _process, address _application, uint _beginDate, uint _dueDate){
        ...
    }

    function requestBerthing( uint _arvDate, uint _dptDate,
                string memory  _mmsi,   string memory _imo) public {

        require(msg.sender == process, "Error sending berthing request.");
        require(_arvDate >= block.timestamp && _dptDate > _arvDate,
                             "Error sending  berthing request.");
        require(!isEmpty(_mmsi), "Error sending berthing request." );
        require(!isEmpty(_imo), "Error sending berthing request." );

        requestList.push( RequestForm(
                        {   requestId: requestCounter++,
                            arriveDate: _arvDate,
                            departureDate: _dptDate,
                            mmsi: _mmsi,
                            imo: _imo,
                            portServices: "",
                            timestamp: block.timestamp,
                            status: Status.WAITING
                        })
                    );
    }
    ...
}
```

Listing 2. Snippet of berthing contract in Solidity.

underflow or re–entrancy and dependency timestamp [ConsenSys Diligence 2020]. As a result, the coding effort is significantly reduced, and so is the number of errors compared to ad–hoc Solidity coding which is known to be error prone. In summary, the abstraction provided by Jabuti can increase code efficiency, clarity, flexibility and reduce the number of coding errors.

6. Conclusion

Companies use a set of applications to support their business management activities. These applications frequently involve external applications owned by third parties. The field of EAI provides methodologies, techniques, and tools for designing integration processes to allow the communication process among these applications. Usually, this communication process needs to comply with some business rules. Therefore, due to some applications belonging to third-party companies, it is essential to monitor this communication process and provide mechanisms to define the rights, obligations, and

prohibitions of the involved parties. Smart contracts can be used for this purpose. However, there is no domain-specific language for writing smart contracts in the EAI domain.

In this paper, we present a domain-specific language called Jabuti to write smart contracts in the application integration domain. Jabuti delivers a high level of abstraction that properly incorporates contract concepts and includes the vocabulary managed in the enterprise application integration domain. Jabuti is user-friendly, enjoys a short learning curve, and produces easy-to-read smart contracts. These features help business experts even with no programming skills to develop their smart contracts. Smart contracts written in Jabuti can be translated into a general-purpose or a cross-domain language.

As a future work, we will implement the translators to automatically convert smart contracts written in Jabuti to Solidity and deploy them on a blockchain. In addition, we intend to develop a monitoring application that will provide the necessary resources to monitor the communication between an application (e.g., Berthing application) and the respective integration process. This monitoring application will capture the information related to the executed operations in the integration process and, on the basis of this information, will trigger the clauses of the corresponding smart contract.

Acknowledgements

This work was supported by the Research Support Foundation of the State of Rio Grande do Sul in Brazil (FAPERGS), under grant 19/2551-0001782-0; the Coordination for the Brazilian Improvement of Higher Education Personnel (CAPES); and, the Brazilian National Council for Scientific and Technological (CNPq) under grant 309315/2020-4. Antonia M. Reina has been funded by projects AETHER-US (PID2020-112540RB-C44/AEI/10.13039/501100011033), COPERNICA (P20_01224) and METAMORFOSIS (US-1381375). Carlos Molina has been partially funded by UKRI CAMB (G115169) project[3].

References

Accord Project (2019). Ergo language guide. `https://docs.accordproject.org/docs/logic-ergo.html`. Last accessed on 15 Dec 2021.

Biryukov, A., Khovratovich, D., and Tikhomirov, S. (2017). Findel: Secure derivative contracts for ethereum. In *Financial Cryptography Workshops*, pages 453–467. Springer.

Bistarelli, S., Faloci, F., and Mori, P. (2021). Towards a graphical DSL for tracing supply chains on blockchain. In *International Workshop on Future Perspective of Decentralized Applications*, pages 1–12. Springer.

ConsenSys Diligence (2020). Ethereum smart contract best practices. https://consensys.github.io/smart-contract-best-practices/attacks/. Last accessed on 01/04/2022.

Dean, D. (2019). Smartfin - implementing a financial domain-specific language for smart contracts. Technical report, Imperial College London.

[3]For the purpose of open access, the author has applied a Creative Commons Attribution (CC BY) licence to any Author Accepted Manuscript version arising from this submission.

Dwivedi, V. K., Norta, A., Wulf, A., Leiding, B., Saxena, S., and Udokwu, C. (2021). A formal specification smart-contract language for legally binding decentralized autonomous organizations. *IEEE Access*, 9(1):76069–76082.

Egelund-Müller, B., Elsman, M., Henglein, F., and Ross, O. (2017). Automated execution of financial contracts on blockchains. *Business & Information Systems Engineering*, 59(6):457–467.

Ethereum Foundation (2021). Solidity documentation. `https://solidity.readthedocs.io/en/latest/`. Last accessed on 5 Jan 2022.

Frantz, C. and Nowostawski, M. (2016). From institutions to code: Towards automated generation of smart contracts. In *2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems*, pages 210–215. IEEE.

Frantz, R. Z., Corchuelo, R., Basto-Fernandes, V., Rosa-Sequeira, F., Roos-Frantz, F., and Arjona, J. L. (2020). A cloud-based integration platform for enterprise application integration: A model-driven engineering approach. *Software - Practice and Experience*, 51(4):824–847.

Frantz, R. Z., Corchuelo, R., and Roos-Frantz, F. (2016). On the design of a maintainable software development kit to implement integration solutions. *Journal of Systems and Software*, 111(1):89–104.

Frantz, R. Z., Reina-Quintero, A. M., and Corchuelo, R. (2011). A domain-specific language to design application integration solutions. *International Journal of Cooperative Information Systems*, 20(2):143–176.

Griffo, C., ao Paulo A. Almeida, J., and Guizzardi, G. (2020). Towards a legal core ontology based on alexy's theory of fundamental rights. In *Multilingual Workshop on Artificial Intelligence and Law*, pages 1–12. CEUR-WS.org.

He, X., Qin, B., Zhu, Y., Chen, X., and Liu, Y. (2018). SPESC: A specification language for smart contracts. In *2018 IEEE 42nd Annual Computer Software and Applications Conference*, pages 132–137. IEEE Computer Society.

Object Management Group (2014). Business Process Model and Notation (BPMN), Version 2.0.2. Technical Report formal/2013-12-09, Object Management Group.

O'Connor, R. (2017). Simplicity: A new language for blockchains. In *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security*, pages 107–120. ACM.

Regnath, E. and Steinhorst, S. (2018). Smaconat: Smart contracts in natural language. In *2018 Forum on Specification Design Languages*, pages 5–16. IEEE.

Seijas, P. L., Nemish, A., Smith, D., and Thompson, S. J. (2020). Marlowe: Implementing and analysing financial contracts on blockchain. In *Financial Cryptography Workshops*, pages 496–511. Springer.

Sharifi, S., Parvizimosaed, A., Amyot, D., Logrippo, L., and Mylopoulos, J. (2020). Symboleo: Towards a specification language for legal contracts. In *2020 IEEE 28th International Requirements Engineering Conference*, pages 364–369. IEEE.

Skotnica, M. and Pergl, R. (2019). Das contract - A visual domain specific language for modeling blockchain smart contracts. In *Advances in Enterprise Engineering XIII*, pages 149–166. Springer.

Skotnica, M., van Kervel, S. J. H., and Pergl, R. (2017). A DEMO machine - A formal foundation for execution of DEMO models. In *Advances in Enterprise Engineering XI*, pages 18–32. Springer.

Szabo, N. (1997). Formalizing and securing relationships on public networks. *First Monday*, 2(9):1–15.

Tateishi, T., Yoshihama, S., Sato, N., and Saito, S. (2019). Automatic smart contract generation using controlled natural language and template. *IBM Journal of Research and Development*, 63(2/3):6:1–6:12.

Varela-Vaca, Á. J. and Reina-Quintero, A. M. (2021). Smart contract languages: A multivocal mapping study. *ACM Computing Surveys*, 54(1):1–38.

Wöhrer, M. and Zdun, U. (2020). Domain specific language for smart contract development. In *2020 IEEE International Conference on Blockchain and Cryptocurrency*, pages 1–9. IEEE.

Wood, G. (2014). Ethereum: A secure decentralised generalised transaction ledger. `https://gavwood.com/paper.pdf`. Last accessed on 20 Jan 2022.